FontAnvil 0.4

Matthew Skala

December 11, 2015

Visit the FontAnvil home page at http://tsukurimashou.osdn.jp/fontanvil.php

FontAnvil user manual
Copyright © 2014, 2015 Matthew Skala

This document is free: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This document is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this document. If not, see http://www.gnu.org/licenses/.

The above license for the document itself notwithstanding, the FontAnvil software described in this document comprises the work of many different copyright holders who have licensed their contributions under a variety of terms. The package as a whole is GPL, but some portions of it are also available under less restrictive licenses. See the "Licensing" chapter of this document for more information.

Anvil clip-art by "Gerald_G," public domain.

Contents

Introduction 4

Building FontAnvil 5
- From a version control checkout . . . 5
- From a distribution package . . . 6
- Special notes for Windows users . . . 6
- Testing . . . 6
- FontAnvil and Tsukurimashou . . . 7

Running FontAnvil 8
- Command-line options . . . 8
- Shebang . . . 9
- Interactive mode and readline . . . 9

Data model 10
- Fonts in memory . . . 10
- Glyphs and slots . . . 10
- Encodings . . . 12
- The `.notdef` glyph . . . 13
- The clipboard . . . 13
- Look-up tables . . . 14
- Syntax, semantics, round trips, and reproducibility . . . 14

The PE Script Language 17
- Basic syntax . . . 17
- Data types, variables, and scope . . . 17
- Operators . . . 18
- Control structures . . . 18

Function and variable reference 19
- Built-in functions in FontForge and not in FontAnvil . . . 32
- Built-in variables . . . 33

Licensing 34

Introduction

FontAnvil is a script language interpreter for manipulating fonts. FontAnvil is substantially compatible with the PfaEdit/FontForge native scripting language, but FontAnvil is intended for non-interactive use; for instance, invocation from the build systems of font packages like Tsukurimashou. To better serve font package build systems in general and Tsukurimashou in particular, FontAnvil has no GUI and, to a reasonable extent, avoids dependencies on external packages.

There was a program called PfaEdit for editing fonts in Postscript ASCII format (`.pfa` files). PfaEdit development continued for many years, it changed its name to FontForge, and it became the *de facto* standard font editing program in the free software community. FontForge is still under active development to this day. The main focus of FontForge is on interactive editing by GUI users, and the proportion of its code and development effort dedicated to such users is large and growing.

PfaEdit had a scripting language, which as far as I know never had an official name. I will call it "PE script" in the interests of neutrality; the traditional filename extension for files in this language is `.pe`. Development of PE script continued into the FontForge era. Many free font packages use PE scripts executed by FontForge to process font files non-interactively in the context of a build system. I myself maintain the Tsukurimashou Project (http://tsukurimashou.osdn.jp/), which processes fonts using PE scripts on a massive scale (thousands of script invocations per build).

Attempting to use a large and steadily-growing GUI program as a non-interactive script language interpreter is not always convenient. The many external libraries needed to build FontForge implicitly become dependencies of any font package that needs FontForge for its build system; and it is not easy to get users to install them all correctly just to build a font package. It is also hard to predict whether a given version of FontForge will actually work for a given script: with GUI enhancements, and even social network features, as the strategic priorities for FontForge development, it is frequently the case that the stable and correct execution of scripts is not at the top of the priority list, and bugs in script processing are fixed late if at all.

FontAnvil is intended to be a stable PE script interpreter for use by the Tsukurimashou Project, to eliminate the dependence on a third-party package whose strategic goals are not closely aligned to Tsukurimashou's. FontForge may well stop supporting PE script entirely in the future; if Tsukurimashou is to survive, then Tsukurimashou cannot be dependent on FontForge.

Only a small amount of effort is likely to be made by the maintainers of FontAnvil and FontForge to retain compatibility with each other. The PE script language is mature and unlikely to change in too many drastic ways, so most scripts written for one interpreter should run correctly on the other for a long time, but already (about one year after the first release of FontAnvil) each interpreter has minor features not supported by the other, and it is likely they will continue to diverge. In this manual, notes on known differences between FontAnvil and FontForge are marked by an *ff* symbol in the margin.

ff

Of all the parts of a forge, an anvil is simple, an anvil is trustworthy, and most of all, *an anvil is stable*.

Matthew Skala
mskala@ansuz.sooke.bc.ca
http://ansuz.sooke.bc.ca/

Building FontAnvil

There are no immediate plans to build binary distribution packages; to use this code you will have to build it yourself from sources.

From a version control checkout

FontAnvil is available by anonymous Subversion checkout from http://svn.osdn.jp/svnroot/tsukurimashou/trunk/. That is the main public repository for FontAnvil source code, and checking out from there is (at least for the moment, while the code is in flux) the preferred way to obtain FontAnvil. The Tsukurimashou repository as a whole is also mirrored on Github at https://github.com/mskala/Tsukurimashou.git.

The version control system does not track some files that would be included in a distribution tarball but can be built automatically from source files already under version control. That includes some parts of the build system, such as "configure," and several source files that are automatically generated by Icemap. *You must build Icemap before you can build FontAnvil from a version control checkout.* If you have checked out the entire project, then Icemap can be found in the `icemap/` subdirectory, sibling to `fontanvil/`.

To go this route you will also need to have all the dependencies of Icemap installed, which include but are not limited to BuDDy (available at https://sourceforge.net/projects/buddy/) and PCRE (available at http://www.pcre.org). It should go without saying that these must be installed in such a way that they *actually work*. I've had reports of Arch Linux, in particular, using a nonstandard dynamic linker configuration such that installing these libraries in the normal way will allow software using them (such as Icemap) to compile but not run. Icemap's configure script will attempt to test for this condition and warn you, but it is not foolproof.

If you wish to build from a version control checkout, it is probably easiest to check out and configure the entire Tsukurimashou Project even if you are primarily interested in FontAnvil. These instructions assume you have done that. It is also assumed you have a complete installation of the standard GNU tools for building C programs, including the gcc compiler, Autoconf, Automake, flex, and so on. Many of these would not be needed for building from a distribution package.

From the root of the Tsukurimashou checkout, run:

```
autoreconf -i
```

This will create configure scripts and other infrastructure for all packages in the project. The `-i` option tells `autoreconf` to automatically create any missing scripts (notably, the one named `missing`) that it thinks it needs.

The `autoreconf` command is likely to produce many error and warning messages, but it should work nonetheless if the Autotools versions are suitable. In particular, note that when `autoreconf` runs it may generate repeated "underquoted definition" warnings in relation to files in /usr/share/aclocal or similar. *This is the fault of some other package on your system;* it is not related to FontAnvil and should be harmless to the FontAnvil build. The issue is that many packages (IMLIB is one common culprit) install M4 files globally to provide customized Autoconf tests, and then if those files are buggy, `autoreconf` complains whenever it is invoked even if, as here, the buggy files are not actually being used.

The top-level Tsukurimashou build will automatically take care of building Icemap and FontAnvil in order to build Tsukurimashou, but assuming you don't want to build the entire project that way, the next step is to build Icemap:

```
cd icemap
./configure
make
```

The configure script supports `--help` and most of the usual options; run that and make appropriate modifications if you wish to customize the Icemap build. Do not ignore error messages from configure.

After building Icemap, you can build FontAnvil. The FontAnvil build will automatically detect

Icemap if it is in the sibling directory created by a source control checkout; it should not be necessary to install Icemap elsewhere first. However, running Icemap requires several files of character-database information from the Unicode Consortium. The `make download` target will automatically go on the Net using `wget` to download them; alternatively, you could read the Makefile to find which files are needed and install them some other way. Thus, the simplest sequence of commands is:

```
cd ../fontanvil
./configure
make download
make
```

At this point FontAnvil should be ready to install with `make install` (root privilege may be necessary for that), or to test with `make check` (test suite is likely to fail at this level of development progress).

From a distribution package

Distribution packages are available from the FontAnvil home page at http://tsukurimashou.osdn.jp/fontanvil.php. Be aware that these packages will *usually* be out of date; and as a consequence of Murphy's Law, it is usual for me to discover many critical bugs immediately after releasing a package.

Building from a package is much the same as building from a version control checkout, minus the need to run Autotools and Icemap. Most of the automatically generated source files are included in the package, so it is not necessary to have Unicode character set files, Icemap, flex, and so on. Unpack the tarball and do the usual Autotools build:

```
./configure
make
# as root:
make install
```

Special notes for Windows users

I cannot realistically offer support for any platforms except Linux and MacOS, which are the ones for which I have access to testing machines. However, FontAnvil's build system is intended to be as portable as realistically possible, and as of shortly before version 0.4, Jeremy Tan has reported some success at compiling FontAnvil under Windows. He has posted unofficial Windows binaries of a patched 0.3 version at http://sourceforge.net/projects/fontforgebuilds/files/fontanvil/ and indicated some willingness to possibly continue updating them. I will also fold back into the mainline any changes that are reasonably unintrusive and help with building on Windows.

Spaces in pathnames, such as "`C:\Program Files`," are a serious problem for all Autotools-based build systems. FontAnvil *probably will not* be able to build if you locate the source code in a path with a space in it. It may have *some*, but not *complete*, ability to handle tools such as compilers and LaTeX that may be located in pathnames with spaces. The `configure` scripts will attempt to detect and warn you about such paths, but be aware that some may cause problems even if `configure` produces no warnings, and the build may be able to compensate even if `configure` does produce warnings.

All I can recommend is to try it and then attempt to work around any problems that occur, by moving things that break into paths that do not include spaces. I'd also like to emphasize that this is a problem with *all* Autotools build systems, not only FontAnvil's. If you want to make a general practice of building Unix software on Windows, the only course of action that will really work is to eliminate filename spaces from your environment. You cannot expect every software package to work around Windows's wacky filenames, and you cannot expect FontAnvil to do so.

After FontAnvil is built there should be no problem with using it in an environment that has spaces in pathnames; this is only an issue for the build.

Testing

FontAnvil includes a test suite, available by running `make check`. Most of the tests are inherited from FontForge, and the test suite had been unmaintained and unused in FontForge for several years before FontAnvil picked it up. As a result, it does not have good coverage, and some of the tests require files that I cannot legally ship. In some cases I have not even been able to identify what the missing files are. If you run the test suite, you should expect many of the tests to be skipped for missing necessary files, and some of them to fail; and even if by some chance you managed to get the entire suite to pass, it is unlikely that that would really mean the software was working properly. Having a better test suite and software that passes it are long-term goals for the project.

After running the tests, you can read the file `tests/coverage.log` to see a summary of how many of the PE script built-in functions were covered by the test suite. As of this writing, it is about 15%.

The option `--enable-valgrind` to `configure` will make the test suite run inside Valgrind, if you have that installed. Valgrind makes the tests much slower, and (with the latest versions as of this writing) it causes at least one of them to fail on Mac OS X through no fault of FontAnvil's, because of Valgrind's lack of support for features used by the Mac system libraries. For these reasons it is not the default. However, if enabled it will produce a lot more information in the log files about the many ways in which FontAnvil abuses memory, and that may be useful in debugging.

The fact that the test suite fails prevents `make distcheck` from working. If you set the environment variable `distcheck_hack` to `0.4` (or the new version number, if I continue using this hack in future versions), then the tests I *expect* to fail will be disabled, so that `make distcheck` can be tricked into accepting the package for distribution. In the long run, this is probably a Bad Thing.

FontAnvil and Tsukurimashou

FontAnvil's reason for existence is to support Tsukurimashou, and its source control repository is a subdirectory of the Tsukurimashou source control repository. FontAnvil does not require Tsukurimashou to build. Tsukurimashou will look for FontAnvil and use it if found, whether installed on the system or inside a subdirectory of its own build. The Tsukurimashou top-level build will also automatically build Icemap and FontAnvil, as needed, if started on a system without them and you don't override this behaviour. Note that this represents a change from earlier version of Tsukurimashou, which required FontAnvil to be built and installed separately first.

All bug reports and other tickets for FontAnvil should be filed through the Tsukurimashou ticket tracker at `http://osdn.jp/projects/tsukurimashou/ticket/`. Set the "Component" field to "FontAnvil."

As a courtesy to Github users, Tsukurimashou's entire source control system (including FontAnvil) is mirrored in my Github account at `https://github.com/mskala`. But osdn.jp remains the authoritative public home of the project. You are welcome to clone the repository—that is why it's there—but the semi-automated gateway from Subversion to Git is one-way. Do not file tickets for FontAnvil on Github.

Do not file tickets for FontAnvil in the FontForge tracker on Github! They are completely separate pieces of software with different maintainers and different goals, notwithstanding the historically shared code and notwithstanding that the FontForge people have graciously given me developer status on their project.

Running FontAnvil

FontAnvil is a script interpreter, so in normal operation it is assumed you already have a script file for it to interpret. Scripts are written in the PE script language described elsewhere in this document. Script files for FontAnvil are traditionally given the filename extension `.pe` (for "PfaEdit"); the extension `.ff` is also popular. Invoking the FontAnvil interpreter then proceeds on more or less the same lines as invoking any other script interpreter.

> FontAnvil's command-line syntax attempts to achieve some degree of FontForge compatibility. However, as of March 2014, FontForge contains at least six different command-line parsers,* and also sometimes hands its command lines off to Python for parsing, so that options interact in complicated ways with each other, with compile-time settings, with operating system shebang support and whether stdin is a terminal or pipe, and so on. FontAnvil does not attempt to match all of this behaviour exactly.

Command-line options

FontAnvil uses GNU `getopt_long_only` to parse command-line arguments; this has the consequence that long options may be specified with `-` (one hyphen) or `--` (two hyphens) as the flag sequence; using `--` is the modern-day Unix convention, but `-` may be preferable for FontForge compatibility. Short options require a single hyphen. Options recognized are as follows.

`-command` ⟨*cmd*⟩, `-c` ⟨*cmd*⟩ Execute a PE script command given literally on the command line. FontAnvil will *not* look for a script file name on the command line if this option is specified; all arguments starting with the first non-option argument become arguments to the script. Only the last invocation of this option will be used; unlike, for instance, Perl, it is not possible to build up a multi-line script by specifying `-c` multiple times.

`-dry`, `-d` Activate a poorly-documented "dry run mode" built into some parts of the FontForge PE script interpreter. This appears to be intended for syntax checking. *Most*, but not necessarily *all*, commands will be skipped. *I do not promise that new code added in FontAnvil will necessarily respect this mode.*

`-help`, `-usage`, `-h` Display a command-line option help message, and terminate without executing a script.

`-lang` ⟨*cmd*⟩, `-l` ⟨*cmd*⟩ Specify interpreter language. If this option is given with the value "`ff`" then it will be ignored for compatibility. Any other value is a fatal error.

`-quiet`, `-q` Reduce the verbosity of error, warning, and informational messages sent to standard error. This option may be specified more than once to reduce the messages even further. Note it may not function exactly like the similarly-named option in FontForge. FontForge often ignores the command-line option and controls message verbosity in some modules by other means including compile-time options, while FontAnvil attempts to unify all verbosity control under this one option.

`-verbose`, `-V` Increase the verbosity of error, warning, and informational messages sent to standard error. This option may be specified more than once to increase the messages even further.

`-nosplash`, `-script`, `-i` Ignored for compatibility.

`-version`, `-v` Display a version and copyright banner, and terminate without executing a script.

`--` Terminate option scanning. All subsequent arguments will be treated as "non-option arguments" (thus eligible to become script file names or script arguments) even if they resemble FontAnvil options. This would be what you might use if for some reason you needed to execute a script file that was named exactly "`-script`."

The first non-option command line argument

*https://github.com/fontforge/fontforge/issues/1277

will be taken as the filename of a script file to execute, unless the `-c` option or one of its synonyms has overridden this behaviour. If the filename so specified is a single hyphen, or if there are no non-option command line arguments at all, then FontAnvil will enter *interactive mode*, reading commands from standard input, as described later in this chapter. Any command line arguments after any script filename will be passed into the script in the variables `$1`, `$2`, and so on—even in the cases of `-c` and interactive mode.

> Option scanning stops at the first non-option argument encountered, which will usually be treated as the script filename. Any arguments after that become arguments to the script (passed in the variables `$1`, `$2`, etc.) and not options for the FontAnvil interpreter. For this reason, options *must* precede the script file name on the FontAnvil command line. For maximum compatibility, the `-script` option should be the last option if you use it at all, with the script file name in a separate argument, not attached using `=`.

Shebang

FontAnvil may be invoked using the shebang convention. Place a line something like "`#!/usr/local/bin/fontanvil`" at the top of a file, and make the file executable, to create a script that can be run like any other program and will automatically use FontAnvil as the interpreter.

Details of shebang support vary depending on the operating system. On most systems, the shebang line must specify an absolute path, and the `env` program may be used to search for a command name in the path to avoid hardcoding the absolute location of the interpreter into a script. There are also special considerations applicable to the length of the interpreter path, arguments specified in the shebang line, and so on.

FontAnvil does not have any special support for shebang. In particular, it does not scan the script to look for its own name in the shebang line. Since the shebang line by definition starts with the comment character `#`, it will be skipped as a comment. FontAnvil just takes the script file name as an argument from the operating system, and (assuming the script name does not happen to be something weird that looks like an option) executes it, with any remaining arguments becoming arguments to the script. This is normally the desired behaviour. However, be aware that it is a technical difference from FontForge, which attempts to determine whether it was invoked via the shebang mechanism and do smart things depending the answer, including working around operating systems that support this feature only poorly. FontForge may possibly *require* options in the shebang line in at least some cases, to select which scripting language it will use.

If you try to specify command-line options in the shebang line, then depending on your operating system's support it is possible that FontAnvil will not see the options even though FontForge would. Some operating systems have unintuitive behaviour regarding options specified in the shebang line; for instance, combining all options into a single string passed as one argument instead of splitting them on spaces. For this reason, authorities on Unix often recommend against using options in the shebang line at all; nonetheless, people continue doing it.

For maximum compatibility with both interpreters, I suggest writing shebang lines in PE script files as you would write them for FontForge (including mentioning the filename "`fontforge`"), and then invoking FontAnvil on the files by other means when desired. That way, FontForge will see the interpreter name and any options it wants, and FontAnvil will ignore them.

Interactive mode and readline

FontAnvil is intended primarily for non-interactive use. However, if it is invoked without a script file name, or with "`-`" (a single hyphen) as the script file name, then it will enter a special *interactive mode*, where it reads commands from standard input and executes them immediately, line by line, rather than reading from a script file. This can be convenient for one-off editing tasks and testing the syntax and behaviour of script commands.

If FontAnvil was compiled with the GNU Readline library and detects that standard input is a terminal, then interactive mode will also offer command-line editing and history using Readline. The usual Readline keystrokes (such as up and down arrows to recall earlier-typed command lines) become available in this mode, and there are some minor changes to the output formatting (in particular, the display of a command prompt) to make it friendlier for interactive users.

Data model

Very many difficulties users have with font editing (both scripted and interactive) come from an incomplete understanding of the data model involved: what entities exist in a font and a font editor and what relationships those entities have with each other. The distinction between glyphs and characters seems to be an especially frequent cause of confusion. This chapter attempts to describe FontAnvil's data model in a way that will be useful to script programmers.

Fonts in memory

Because PE script was originally designed for controlling a GUI font editor, it treats fonts as documents to be opened and closed, much as a GUI editor might.

At any given time there exists a global set of fonts that are *open*. These are stored in RAM. Open fonts are associated with filenames, even if the filenames do not actually exist on disk; the interpreter will assign temporary filenames (usually similar to "`Untitled1.sfd`") to fonts that were created in memory and not loaded from files. The filenames should be unique (no more than one open font sharing a filename), and it's not easy to create a situation where they are non-unique, but I suspect that having distinct open fonts with duplicate filenames may be technically possible and likely to trigger bugs if attempted.

The set of open fonts is technically a sequence (with a specific order), not an unordered set, and the order is visible to scripts through the `$firstfont` and `$nextfont` built-in variables, but this fact is seldom important.

At most one of the open fonts may be the *current font*. This state is global. Most font-editing operations implicitly apply to the current font. The `Open()` built-in function sets the current font, but its exact behaviour is context-sensitive. If the specified filename is already an open font, then `Open()` just sets the current font to that one. If the specified filename is *not* already an open font, then `Open()` loads it from disk, causing it to become an open font, before setting the current font to that one.

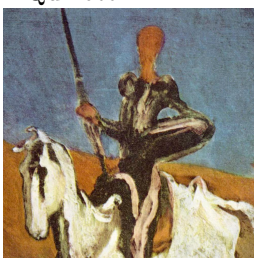
When the interepreter starts up, the set of open fonts is empty and there is no current font. In this state, any operation that implicitly refers to the current font will fail; scripts can only use a small subset of the language to create or open a font and make it current. The `Close()` built-in function removes the current font from the set of open fonts (without saving it to disk—that must be done as a separate operation if saving is desired) and places the interpreter back into the state of having no current font.

The existence of a no-font state is a difference between PE script interpreters (both FontAnvil and FontForge when run as a command-line interpreter) and the FontForge GUI. The GUI insists on always having at least one open font and always having a current font, enforcing this rule by automatically loading fonts from earlier editing sessions, automatically creating a new empty font if the load fails, choosing another open font to be current when one is closed, and terminating the program when the last font is closed.

ff

Glyphs and slots

Here are two pictures of Don Quixote.*



The pictures are different, but they are pictures of the same fictional character. In the same way, we can have several pictures of the same character in a writing system.

a *a* **a**

What these three "a"s share is the *character*; what they do not share is the *glyph*. Fonts contain

*Left: Gustave Doré, 1863. Right: Honoré Daumier, 1868.

collections of glyphs, concrete things, which are pictures of characters, abstract things. It is often the case that in any given font, each glyph corresponds to exactly one character and vice versa; but there are many important exceptions to that rule. To understand how FontAnvil processes fonts it is important to bear in mind that fonts are collections of glyphs, and glyphs are not the same thing as characters despite being closely connected with characters.

This glyph (the classic ff-ligature) is not associated with one character, but with a sequence of two characters:

There is no double-f character, as a separate abstract entity distinct from just two ordinary "f"s in a row, in the English language. There is no standard character code for such a thing.[†] Nonetheless a font for high-quality typesetting of English must contain a glyph for this double-f entity that is not quite a character. Despite such exceptions, one glyph per character is true most of the time in English. In some other languages, the conceit of glyph-character equivalence breaks down entirely. In Arabic, for example, many characters have four or more visual forms requiring separate glyphs in a font, depending on how they connect to neighbouring characters.

FontAnvil represents a font in memory as including a zero-based array of *glyph slots*. The array is variable-sized, but it is a true array data structure, not a list: all slots from index 0 up to one less than the number of slots exist as long as the in-memory font does. Creating a new glyph means overwriting the possibly-blank previous contents of some slot. Destroying a glyph means filling the slot with a blank glyph, but the slot continues to exist. Changing the number of slots can only be done by increasing or decreasing the length of the array, and encodings (described in the next section) may constrain the number of slots.

Glyph slots in a font always have *glyph numbers* which are their indices in the array. Glyphs in a font cannot meaningfully be said to be in any specific order other than the order determined by the array indices. Every glyph has a number, and every number has a glyph slot. No two glyphs can have the same slot; two slots may contain identical glyphs, or even a "reference" from one to the other, but the two slots' contents will not truly be the same entity. A glyph slot might be *blank*, that is devoid of outlines and other data, and people usually think of such slots as not really being glyphs. But some attributes of a glyph slot (such as its name) are required to always have non-null values, even on blank slots.

> FontAnvil's in-memory glyph slots can be blank but never truly empty. However, most on-disk file formats *do* have a concept of a glyph failing to exist at all. Blank slots are usually not written when saving from memory to disk, and loading a file from disk to memory that does not fill all slots will usually leave the others blank.

Every open font has a *selection*, which specifies a set of the glyph slots (more about glyph slots in the next section). Many editing operations implicitly operate on the selection of the current font. Although every open font has a selection, usually only the selection of the current font is relevant. Open fonts retain their selections through changes in which open font is current.

The selection is actually a sequence, not a set, of glyph slots. That means the slots in it can be selected in a specific order. This distinction is relevant in the FontForge GUI, where you can select several glyphs in a specific order, open a "Metrics" window, and have them come up in the order you selected them. However, each glyph slot can appear at most once in the selection, and the order of the selection is seldom if ever observable or controllable from the scripting language. It is usually better to think of it as a set with no specific order, not as a sequence.

The selection is applied at the level of glyph slots. It is perfectly possible for the selection to include blank glyph slots, because it is defined as a set of slots, not a set of non-blank glyphs. Nonetheless, one often only cares about the non-blank glyphs, and some commands for manipulating the selection will automatically limit themselves to slots that are not blank.

Glyph slot numbers *may or may not* be closely connected to Unicode, ASCII, or other character codes, depending on issues discussed in the next section. It is important to be aware that glyph slot numbers are not the same type of entity as charac-

[†]Actually, there is one in Unicode, but you're not supposed to really use it; the details of that are beyond the scope of this discussion.

ter codes, despite sometimes having equal numerical values.

Encodings

Fonts do not exist solely to be edited with FontAnvil. To be useful, a font must eventually be saved in some format understandable by a word processor or similar application; and the external software must be able to associate the glyphs in the font with the characters in text that will be typeset using the font. There will necessarily exist some *code* that associates numbers called *code points* with the different characters that might appear in text, and a font file must explicitly or implicitly describe which code points go with which glyphs. It is worth emphasizing that code points refer to characters, *not glyphs*.

The ASCII code is familiar to many computer users, especially in the English-speaking world, but is inadequate for languages other than English. Today, nearly everybody uses a code called Unicode. Unicode's code points coincide with ASCII for all the characters covered by ASCII; but it also covers many thousands of other characters. It is supposed to be a universal code for all text in all human languages. But Unicode did not always exist and its use was not always universal. Most common font formats are designed to also accomodate character encoding schemes predating Unicode or simply other than Unicode. Frequently, a font file will include translation mappings for several different codes, in the hope that software using the font can find its own preferred code among the choices. There needs to be a way to translate code points (which identify characters) into glyph numbers (which identify glyphs), even in cases like ligatures and variant glyphs where this translation is more complicated than a simple one-to-one lookup.

FontAnvil includes several mechanisms for addressing these issues, but the most significant is that of the *encoding*. The encoding is a property of each font (global to the font) and describes a set of assumptions and constraints about the relationships between code points and glyph numbers.

> Every glyph slot in a font always has a glyph number, no two slots can have the same number, and the set of glyph numbers that exist is always the set of integers from 0 up to one less than the number of glyphs in the font. Every code point designates at most one glyph slot. But a glyph slot might have zero, one, or more than one code point; a code point might have no glyph slot; and the set of code points that exist might not be a simple interval of the integers. Nonetheless, the font's encoding may trigger the enforcement of constraints that make the code point situation less complicated.

Most of the possible values of the encoding field are associated with standardized character codes defined by external organizations. Each code defines a meaning for a range of code points from 0 up to some number that depends on the code. *When the encoding is associated with an externally standardized code other than Unicode*, FontAnvil enforces the following constraints:

- There must be at least as many glyph slots as there are code points in the code.
- Glyph slots in the range 0 through one less than the number of code points in the code correspond one-to-one with the code points.
- Any glyph slots with greater indices have no code points.

The case of *unencoded glyphs*, those in glyph slots beyond the end of the code point range specified by the encoding, is important. Glyphs like the ff-ligature mentioned earlier, or the alternate forms of letters in a script like Arabic, usually fall into this category. When a word processor typesets text using a font, it starts out by translating the code points into glyphs one-for-one according to the basic code points of the glyph. The result of that translation cannot contain unencoded glyphs. But the basic one-for-one translation of code points to glyphs is only a starting point. Further processes can merge and split glyphs so that more than one character can be typeset with one glyph, one character can be typeset with more than one glyph, and which glyph goes with which character can be different in different contexts. These further processes can bring unencoded glyphs into play. The encoding does not specify the number of unencoded glyph slots that may exist after the range of encoded glyphs. The unencoded glyph slots may be manipulated by built-in functions like `SetCharCnt()`; encoded glyph slots may not be added or removed.

Each glyph slot has a property called the *Unicode number*. This is a code point (in the code that is named Unicode), but I am going to call the num-

ber in this field just a "number" to distinguish it from the code points that exist in non-Unicode encodings. When the encoding is one of the standardized non-Unicode encodings, the constraint is enforced that the Unicode number must be the correct translation (using the iconv library) of the glyph number for encoded glyphs, or the null value of -1 for unencoded glyphs. For example, if the font's encoding is KOI8-R (commonly used for Russian text), then glyph slot number 241 is for the letter "ya," which looks like a backwards R. FontAnvil will enforce the constraint that this slot's Unicode number is 0x042F, which is the Unicode code point for that letter. "The constraint is enforced" means that if you try to change the value of the Unicode number field, the font's encoding will be immediately changed to Custom. Editing the Unicode numbers is not compatible with keeping the encoding and its fixed mapping.

But not every value for the font's global encoding field is associated with an external standard other than Unicode. When the font's encoding field refers to some form of Unicode, or does not refer to an external standard, then additional special considerations apply; and in fact, these special cases are the most popular and useful values for the encoding field.

When the encoding is set to Custom, few encoding-related constraints are enforced. There may be any number of glyph slots. Any slot may have a Unicode number, or not, and there is not necessarily any relationship between the Unicode numbers and the glyph slots.

There are two Unicode encoding options, Unicode (BMP) and Unicode (Full). These each behave more or less like the non-Unicode standardized encodings. One difference is that it appears sometimes possible to set the Unicode number of a glyph slot such that it does not match its glyph number. This may be a bug. FIXME investigate further - this description may possibly be simplified if Unicode and non-Unicode turn out to really behave the same.

FIXME investigate and document the "Original" encoding option.

Glyph slots have names. All glyph slots have non-empty names, including blank slots, and all names must be unique within a font. The names are sometimes automatically assigned and may also be manipulated by script commands; but constraints (including the requirement for uniqueness) will be enforced on such manipulation.

> People who think they want to edit glyph slot names are often wrong.

FIXME document name lists

The `.notdef` glyph

FIXME

The clipboard

There is a global entity called the *clipboard*, which holds glyph data of the kind that might be stored in glyph slots, such as outlines, anchors, and references. The clipboard is like a font in that it can store a bunch of slots' worth of data, in a definite order, but the clipboard is unlike a font in that the slots do not have meaningful numbers and it does not store slot attributes other than glyph data, such as slot names and code points.

The usual way of using the clipboard is somewhat like using the clipboard in any common GUI document editor: select some slots, do a cut or copy operation, select some other slots (even in a different font), and do a paste operation. Here is typical code to copy the uppercase ASCII alphabet from an existing font into a new font (leaving many things in the new font empty or default, which may cause problems later):

```
Open("font1.sfd");
Select('A','Z');
Copy();
New();
Select('A','Z');
Paste();
Save("font2.sfd");
```

One thing to be aware of is that `Paste()` always writes into the selection, and so you must create a nonempty selection for `Paste()` to be meaningful. This differs from a word processor that can "insert" text; FontAnvil treats a font as fixed framework of glyph slots that can only be changed by overwriting. Inserting or deleting in the middle, in a way that changes the number of slots that exist, would disrupt the framework of the encoding and is rarely, if ever, what you really want.

> A glyph slot's name is associated with the glyph slot, not with the glyph data stored in the slot. The slot name will not move with the glyph data when the glyph data is cut and pasted into a new slot. Unicode code points, and any other

encoding numbers, are also parts of the glyph slot and will not move with cut and pasted glyph data.

Look-up tables

FIXME

Syntax, semantics, round trips, and reproducibility

It is a very common complaint from users of font editors that someone loaded a file, saved it without making any other changes, and "the result was not the same." Such complaints are often filed as bugs in the FontForge bug tracker. Dealing with them is not so easy as it might appear on the surface, because what it means for two files to be "the same" is not so simple as whether they contain the same sequence of bytes. This section is intended to clarify some of the relevant issues as they relate to FontAnvil, and let users know what are and are not reasonable expectations about the system's behaviour.

Consider the two ASCII strings "`1.5`" and "`1.50000000000`"; are those the same? As strings, they are not. But if these strings are representations of a real number, they represent the same real number. So if an editor loads one and saves the other, the bytes in the file will have changed but the number will not have changed. We can say that the *syntax* has changed but the *semantics* have not. These are terms borrowed from the field of linguistics, where they describe different levels of abstraction in the study of a language: syntax refers to questions like what sequences of words are grammatically valid sentences, while semantics refers to the meanings of words.

Note that if ASCII strings represent floating point numbers in low machine precision, we might even have "`1.50000000000`" and "`1.49999999999`" referring to the same number. The "number" that matters is the machine-precision binary number; two strings that both result in the same machine-precision binary number are two strings with the same meaning even if a human translating them instead to exact real numbers would see a difference.

When FontAnvil, or any other font editor, loads a file from disk, the file is converted into an in-memory representation that is designed to store the semantics of the font. When it saves the file, that in-memory representation is converted into the disk file format. As a result of this process, in general it is reasonable to expect that loading and saving a file will preserve the semantics of the data in the file, but in general *it is not reasonable to expect aspects of syntax beyond semantics to be preserved.*

The result of a round-trip through any editor will in general be a file with the same semantics, but not necessarily the same syntax. In general that statement is true of all software that edits complicated file formats, including word processors and image editors as well as all font editors, not only FontAnvil. For reasons unknown to me, users of font editors seem especially inclined to be surprised by the loss of non-semantic information and to report it as a claimed "bug," even though this is the normal, expected, and pretty much inevitable behaviour of most computer software.

Further confusion may ensue when a file is round-tripped through more than one on-disk format, because different formats may define different semantics, sometimes at the ontological level; that is, different formats may be based on different models of what are the important concepts in the problem domain. For instance, some font formats make heavy use of "references," which allow part of one glyph to automatically incorporate another glyph. Other font formats just do not contain references at all. Some font formats use quadratic splines to represent curves, while others use cubic splines, and each of these is only approximately capable of representing data from the other; a loss of precision necessarily occurs when converting between them. In general, information from one format may need to be changed in irreversible semantic ways or even discarded in order to convert a font to another format. Converting back and getting even the same semantics, let alone the same syntax, is not always a reasonable expectation. This issue is somewhat analogous to what occurs when translating text successively through two or more very different human languages.

The FontAnvil in-memory format attempts to be able to represent all the semantic information in all the on-disk formats that FontAnvil supports; but it still represents semantics only. Since FontAnvil descends from FontForge which descends from PfaEdit which was originally an editor for Postscript only, the FontAnvil in-memory format may have some bias toward a Postscript data model. It nonetheless is intended to cover the semantics of all the formats FontAnvil supports.

The FontAnvil SFD "native" format is a special on-disk format intended to represent the semantics of the in-memory format as accurately as possible; so, by design, it should be possible to round-trip between a font in memory and a font on disk in SFD format, with no loss of semantics. However, it is still possible to find syntactic distinctions in SFD files that will not be preserved by loading and saving. For instance, one can mess around with external software to re-arrange the sequence of glyphs in an on-disk SFD file, only to have one's tampering automatically undone the next time the file is loaded and saved because the order of entries in the file is not semantic and all that really counts is the glyph numbers stored in the appropriate fields.

From this general picture there emerge several points that describe how FontAnvil relates to the syntax and semantics of font files.

- FontAnvil attempts to preserve semantics of font files through a round trip of loading from and saving to any one supported foreign format.
- FontAnvil attempts to preserve semantics of font files through a round trip of loading a foreign format, saving and loading the data in SFD format, and then saving back to the same foreign format.
- When loading from one foreign format and saving to another, FontAnvil attempts to preserve whatever semantics make sense in both formats; but a round trip from one foreign format to another and back may not be lossless.
- FontAnvil does not attempt to preserve syntax of any file format beyond the semantics of the font inside, and in general it should be expected that it usually *will* change syntax.
- FontAnvil does not attempt to preserve SFD file semantics specific to features FontForge includes and FontAnvil does not; for instance, pickled Python objects.
- FontAnvil does not support SFD-like files written or edited by software other than FontAnvil itself, except versions of FontForge close to the Git repository version current as of March 2014.
- FontAnvil does not support FontForge SFDir (SFD split into a directory instead of a single file) format.

The Debian Project has recently embarked on a misguided project to demand "reproducibility" of everything in their world, which they define as having the ability to compile software many times under different circumstances and get byte for byte syntactically identical output. I call this *extreme reproducibility* to allow distinguishing it from weaker properties that might also be called reproducibility, and to emphasize what an outlandish and cumbersome form this one actually is. Several purported reasons for extreme reproducibility to be desirable are put forward by the project. The claim that extreme reproducibility would be nice for security is true, because multiple parties can independently compile something, check that they got the same result, and thus be sure that they are all either uncompromised or compromised in the same way. But not everything that would be nice to have is worth what it costs.

Absent rigorous audits of everything else that can go wrong, an extreme reproducibility demand is far from a guarantee of security. Demanding extreme reproducibility precludes embedding metadata about the circumstances of a build in the built file; as soon as there's so much as a timestamp inside a file, byte for byte comparisons break. Metadata is especially important and useful in fonts and typesetting; and third-party file formats in this application domain often have specific requirements for timestamps and even per-save random ID numbers to be included in the files, making it difficult to obey the file format definitions while also having extreme reproducibility.

Randomized algorithms, and computations in which multiple threads work simultaneously, may often generate differing but in some sense close enough results when run more than once; anything of that nature is forbidden by extreme reproducibility. Cryptographic signature algorithms often require the opposite of extreme reproducibility, with one-time nonce values that (on pain of destroying the security guarantees) must *never* be the same from one run to the next; any part of a build process that involves such a signature must be carefully segregated from any parts that are attempting to guarantee extreme reproducibility.

With all that in mind, FontAnvil rejects the extreme form of reproducibility, but embraces the weaker common-sense reproducibility principle that running the same process more than once should produce functionally identical results.

- FontAnvil scripts are intended to be *semantically* reproducible: running the same script on the same inputs with the same FontAnvil

binary should produce output with the same meaning.

- It is not the policy of the FontAnvil project to make any attempt at *syntactic* reproducibility, it is not reasonable to expect output to be byte for byte identical from one run to another, and in general, the contrary should be expected.

The PE Script Language

I did not invent the PE scripting language, and the person who did never fully specified or documented it. This documentation is based partly on reverse engineering; is descriptive, not prescriptive; and may not be complete, nor even correct as far as it goes. The only way to be sure what a PE script will really do is to run it and find out, like Rikki-Tikki-Tavi.

Basic syntax

Scripts are text files. The traditional filename extension is `.pe` ; scripts in the wild have also been seen using a `.ff` extension.

Comments may be marked in any of these ways:

```
# hash for shell-like line comment

// two slashes for C++-like line comment

/*
C-style comment delimiters,
which may cover multiple lines.
*/
```

Newlines are syntactically significant, marking the ends of statements. To continue a statement onto more than one line, you must use a backslash to escape the newline.

ff

FontForge processes line-continuation backslashes in a separate abstraction layer before the main interpreter sees them, which causes some effects that language users may not expect. For instance, in FontForge a line-based comment started by `#` or `//` may be continued onto a new line by a backslash, without any `#` or `//` on the continuation line. FontAnvil instead processes line-continuation backslashes in the main tokenizer. They may not be used to continue a line-based comment onto a new line, and they may not be used in the middle of a token (such as a function name) without having the effect of splitting it into two tokens. However, they may be used within string constants.

Semicolons also mark the ends of statements, and may be used to join multiple statements onto a single line. Semicolons at the ends of lines create empty statements, which are ignored.

PE script is case sensitive for reserved words, variable names, and built-in function names.

Data types, variables, and scope

Values have associated types. Variables can hold values of arbitrary type and remember what type they are. The types are:

- integer
- floating-point number
- Unicode code point (note that this is a distinct data type from "integer")
- string (of bytes, usually assumed to be UTF-8; null terminated)
- array
- void

Syntax for constant values looks like this:

```
# integers in decimal, hexadecimal, or
# octal, using C syntax
123     # first digit nonzero for decimal
0x52    # 0x prefix for hex
041     # 0 prefix but not hex means octal

# floating-point numbers indicated by the
# decimal point; note the decimal point
# is always . regardless of locale
123.45  # basic decimal float
4.9e5   # scientific notation, = 490000.0

# Unicode code points are hexadecimal
# numbers marked by 0u
0u1f4a9 # everybody's favourite

# strings have single or double quotes
'Single'
"double"
"foo\nbar" # \n for newline, in both
"foo\
```

```
bar" # backslash continues string past
     # a line break, this is "foobar"
"foo\\bar" # backslash makes other chars
           # literal, this is "foo\bar"
'"' # same quote starts and ends string

# unescaped newline also validly ends a
# string, but don't do this!
# current FontAnvil supports it for
# FontForge compatibility only
"foo

# square brackets and commas for arrays
[1,2,3,0uABC,'foo']
```

Literal string constants in PE script syntax are limited to 256 characters. You can, however, construct longer strings with multiple literals and the concatenation operator.

The language seems intended to allow arrays to have more than one dimension (i.e. each element of an array may itself be an array) but such arrays are currently broken in both FontForge and FontAnvil, and usually cause the interpreter to crash. I hope to fix this bug in FontAnvil, but if I fix it and FontForge doesn't, then any scripts that make use of multidimensional arrays will be incompatible with FontForge.

Operators

FIXME

Control structures

FIXME

Function and variable reference

ATan2

`ATan2(`*y*, *x*`)`

Returns the arctangent of y/x in radians, using the signs of the arguments to choose the quadrant. *Note the order of the arguments, with y first.* This function mimics the behaviour of the C math library `atan2()` function. This function may be used without a loaded font.

AddAccent

`AddAccent(`*accent*, [*pos-flags*]`)`

Perform one step in the process of building a composite glyph from a base and an accent. This involves three glyph slots:

- The base glyph, which might be a letter, like e. This must exist somewhere in the font.
- The glyph slot that will contain the composite, which might be an accented-letter glyph like ë. When `AddAccent()` is called, this slot must be selected and must be the only thing selected. It must already contain a reference to the base glyph, and that must be the first reference.
- The glyph slot containing the accent itself. This must be specified as the first argument to `AddAccent()`, either as a string which names the accent glyph slot or as a Unicode code point.

FontAnvil will attempt to place the accent in the composite glyph slot, in an appropriate position relative to the base. Without the second argument, it will choose that position depending on the Unicode value of the accent. The second argument if specified may be one of the following integer flags. In principle, it could be a bitwise OR of more than one of them, but that makes very little sense.

flag	position
0x100	Above
0x200	Below
0x400	Overstrike
0x800	Left
0x1000	Right
0x4000	Center Left
0x8000	Center Right
0x10000	Centered Outside
0x20000	Outside
0x40000	Left Edge
0x80000	Right Edge
0x100000	Touching

AddAnchorClass

`AddAnchorClass(`*name*, *type*, *subtable*`)`

Add an anchor class (for use with OpenType GPOS features) to the current font. All three arguments are required, and are strings. The first is the name of the class; the second is one of `"default"`, `"mk-mk"`, or `"cursive"`, indicating the general kind of attachment (mark to base, mark to mark, or cursive, respectively); and the third is the name of the lookup subtable in which the class will be used.

AddAnchorPoint

`AddAnchorPoint(`*name*, *type*, *x*, *y*, [*lig-index*]`)`

Add an anchor point for mark attachment to the currently selected glyph. It is an error to attempt this if no, or more than one, glyph is selected. The *name* argument specifies the class, such as was used in *AddAnchorClass()*. The type should be one of `"mark"`, `"basechar"` (syn. `"base"`), `"baselig"` (syn. `"ligature"`), `"basemark"`, `"cursentry"` (syn. `"entry"`, `"cursexit"` (syn. `"exit"`), or `"default"`. The `"default"` choice attempts to guess an appropriate type. The x and y arguments specify the coordinates of the anchor. The final argument is required if and only if the type is base ligature.

AddDHint

`AddDHint(`x_1, y_1, x_2, y_2, x_u, y_u`)`

Add a diagonal hint to any selected glyphs. A diagonal hint is specified by three pairs of X-Y coordinates: (x_1, y_1) and (x_2, y_2), which specify two points on opposite sides of the diagonal stem, and (x_u, y_u), which specifies a unit vector in the direction of the stem.

AddExtrema

`AddExtrema(`[*really*]`)`

Add extra points to spline segments (that is, break them into smaller segments) in the selected glyph slots to ensure that segments achieve their maximum and minimum X and Y values at endpoints and not in between. This is a technical requirement of some font formats.

If an extremum occurs very near but not at the endpoint of a segment, then `AddExtrema()` will by default skip processing that extremum in order to avoid creating a very short segment. Specify a nonzero integer for the optional argument *really* to force adding extrema in such cases.

Rounding coordinates in a font to integer values will often cause the splines to break the points-at-extrema rule, and adding points at extrema will often break the integer-coordinates rule required by the same font formats! I do not know of a sequence of rounding, extrema-adding, and simplification operations without manual intervention that will reliably bring a font into compliance with both rules.

AddHHint

`AddHHint(`*start*,*width*`)`

Add a horizontal hint to any selected glyphs, starting at X coordinate *start* and extending for distance *width*.

AddInstrs

`AddInstrs(`*dest*, *replace*, *instrs*`)`

Add Microsoft-style rasterization hints ("instructions") to the font. These can be stored either in a TrueType table, or in an individual glyph, as determined by the string argument *dest*. If *dest* is `"fpgm"` or `"prep"`, then the instructions will go in the table of that name; if it is an empty string then the instructions will go in any selected characters; and otherwise, the string value will be taken as the name of the glyph where the instructions will go. Glyphs actually named "fpgm" or "prep" apparently can only be specified by the selection mechanism.

The *replace* argument should be an integer; if it is nonzero then the new instructions replace any currently in the specified destination, and if it is zero then they are appended to any already present. The *instrs* argument should be a string containing the instructions. This is code in a primitive assembly language for the TrueType instructions bytecode interpreter. The syntax defined by the TrueType standard appears to be supported, with some undocumented but optional extensions proprietary to FontForge (and inherited by FontAnvil).

AddLookup

`AddLookup(`*arg*, *arg*, *arg*, *arg*, *arg*`)`

AddLookupSubtable

`AddLookupSubtable(`*arg*, *arg*, *arg*`)`

AddPosSub

`AddPosSub(`*arg*, *arg*, *arg*, [*arg*], [*arg*], [*arg*], [*arg*], [*arg*], [*arg*], [*arg*]`)`

AddSizeFeature

`AddSizeFeature(`*arg*, *arg*, [*arg*], [*arg*], [*arg*]`)`

AddVHint

`AddVHint(`*start*,*width*`)`

Add a vertical hint to any selected glyphs, starting at Y coordinate *start* and extending for distance *width*.

ApplySubstitution

`ApplySubstitution(`*script*,*lang*,*feature*`)`

Apply OpenType-style substitutions to selected glyphs. All three arguments should be strings of up to four characters, which will be blank-padded if shorter; the *script* and *lang* tags (but not *feature*) may also have the wildcard value * (a single asterisk), which matches everything. FontAnvil will apply any substitution rules in the current font for the selected glyphs that match the specified script, language, and feature (or ignoring the checks for script or language if applying the wildcard); glyph slots for which a substitution is found will be replaced with the contents of the slots specified by the substitution rules.

Context-dependent substitutions appear to be ignored, and it is not specified what happens if the result of the subsitution rule is not exactly one glyph.

Array

`Array(`*size*`)`

Creates and returns a value of array type, with the given number of elements initialized to void. This function may be used without a loaded font.

AskUser

`AskUser(`*prompt*, [*default*]`)`

Asks the user a question. The string *prompt* is written out to standard output, and the next line from standard input is captured to become the return value. A line is terminated by a newline character, which will be included in the return value. On error, or if the user enters a blank line, the return value will be *default* if specified, or an empty string if not. This function may be used without a loaded font.

ff

If FontAnvil was compiled with readline support, then `AskUser()` will allow command-line editing. This is a FontAnvil extension, not supported by FontForge.

AutoCounter

`AutoCounter()`

AutoHint

`AutoHint()`

Automatically generate Adobe-style rasterization hints for selected glyphs.

AutoInstr

`AutoInstr()`

Automatically generate Microsoft-style rasterization hints (that is, "instructions") for selected glyphs.

AutoKern

`AutoKern(`*arg*, *arg*, *arg*, *arg*`)`

AutoTrace

`AutoTrace(`, ...`)`

AutoWidth

`AutoWidth(`*arg*, *arg*, [*arg*]`)`

BitmapsAvail

`BitmapsAvail(` *sizes*, [*rasterized*]...`)`

Choose the set of bitmap or greymap sizes ("strikes") that will be stored in the current font. The *sizes* argument should be a list of integers; these are either plain numbers of pixels (implicitly specifying one bit per pixel black and white bitmaps); or bit-field encoded numbers specifying both a pixel size (in the least significant 16 bits) and a number of bits per pixel (in the higher bits). For instance, 0x8000C specifies a 12 pixel greymap font with eight bits per pixel.

If any sizes are specified that are not already in the font, they will be added. If any sizes are not specified but exist in the font, they will be removed; so the final list of sizes will match what was specified. Any newly-created strikes will be filled in with rasterized versions of the splines in the font, if *rasterized* is not specified or is a nonzero integer; the new strikes will be blank if *rasterized* is zero.

BitmapsRegen

`BitmapsRegen(` *sizes*`)`

Render bitmaps from the splines in the current font. The list of bitmap sizes (and bit depths, in the case of greymap strikes) is specified as an array in the same format as with `BitmapsAvail()`. All the specified sizes must already exist. Their contents will be replaced with the results of the rendering; any other existing sizes are not changed.

BuildAccented

`BuildAccented()`

If any selected glyphs are accented letters (determined by Unicode code points, according to George Williams's undocumented rules), then replace them with composite glyphs formed by references to base and accent glyphs, in an unspecified way.

BuildComposite

`BuildComposite()`

If any selected glyphs fall into a category described as "ligatures and whatnot" (determined by Unicode code points, according to George Williams's undocumented rules), then replace them with composite glyphs formed by references to other glyphs, in an unspecified way.

BuildDuplicate

`BuildDuplicate()`

For all selected glyph slots, if the glyph slot is associated with a Unicode code point that is an alternate of some other Unicode code point according to undocumented rules, and there is a glyph in the other code point's slot, then point the current glyph slot to that other glyph, thereby creating a deprecated multi-slot glyph structure. What happens to

any former glyph in the current slot is not clear; the code looks like it may leak the memory.

The intended use of this function seems to have been mainly to build fonts with backward compatibility to certain Chinese encodings (perhaps based on early versions of Unicode without full coverage) that placed glyphs in the PUA for characters that are also (now) found at standard code points.

Do not use this function. It is kept in FontAnvil (for the moment, quite possibly to be removed some day) for backward compatibility, because some scripts may depend on the built-in rules. Because they are undocumented, those are not easy to reproduce with other functions. Creating multi-slot glyph structures in general is deprecated in the relevant font standards and so should rarely be attempted; and if you really need a multi-slot glyph structure, the `SameGlyphAs()` function is a more controllable way to build them.

CIDChangeSubFont

`CIDChangeSubFont(`*arg*`)`

CIDFlatten

`CIDFlatten()`

CIDFlattenByCMap

`CIDFlattenByCMap(`*arg*`)`

CIDSetFontNames

`CIDSetFontNames(`*arg*, *arg*, [*arg*], [*arg*], [*arg*], [*arg*]`)`

CanonicalContours

`CanonicalContours()`

CanonicalStart

`CanonicalStart()`

Ceil

`Ceil(`x`)`

Returns $\lceil x \rceil$. That is the ceiling of x, or the least integer greater than or equal to x. This function may be used without a loaded font.

CenterInWidth

`CenterInWidth()`

ChangePrivateEntry

`ChangePrivateEntry(`*arg*, *arg*`)`

ChangeWeight

`ChangeWeight(`[*arg*]`)`

CharCnt

`CharCnt()`

Return the number of glyph slots in the current font, including both encoded and unencoded slots.

CharInfo

`CharInfo(`*arg*, *arg*`)`

CheckForAnchorClass

`CheckForAnchorClass(`*arg*`)`

Chr

`Chr(`*int*`)`

Takes a single integer in the range -128 to 255 and returns a string containing that byte, folding negative values into two's complement notation.

`Chr(`*array*`)`

Takes an array of integers in the range -128 to 255 and returns a string consisting of those bytes. This is the inverse of the `Ord()` function; see also `Utf8()` and `Ucs4()`, which operate on Unicode strings and code points instead of byte values.

This function may be used without a loaded font. Note that the integers are *byte* values. Code points U+0080 and beyond may be constructed by spelling them out in UTF-8. It is also possible, but not recommended, to construct strings that are invalid UTF-8.

Since strings are stored in null-terminated form internally, passing a zero, although not reported as an error, will cause the string to terminate immediately before the zero. It is not possible to create a string value in PE Script that meaningfully *contains* a zero byte.

I have submitted a patch to FontForge, but as of the current writing (June 2015), FontForge does not document its support of zero and negative numbers, and documents but does not correctly implement array-valued arguments. FontAnvil behaves as documented here.

Clear

`Clear()`

ClearBackground

`ClearBackground()`

ClearCharCounterMasks
`ClearCharCounterMasks()`

ClearGlyphCounterMasks
`ClearGlyphCounterMasks()`

ClearHints
`ClearHints([`*arg*`])`

ClearInstrs
`ClearInstrs()`

ClearPrivateEntry
`ClearPrivateEntry(`*arg*`)`

ClearTable
`ClearTable(`*arg*`)`

Close
`Close()`

CompareFonts
`CompareFonts(`*arg*, *arg*, *arg*`)`

CompareGlyphs
`CompareGlyphs([`*arg*`], [`*arg*`], [`*arg*`], [`*arg*`], [`*arg*`], [`*arg*`])`

ControlAfmLigatureOutput
`ControlAfmLigatureOutput(`*arg*, *arg*`)`

ConvertByCMap
`ConvertByCMap(`*arg*`)`

ConvertToCID
`ConvertToCID(`*arg*, *arg*, *arg*`)`

Copy
`Copy()`

CopyAnchors
`CopyAnchors()`

CopyFgToBg
`CopyFgToBg()`

CopyGlyphFeatures
`CopyGlyphFeatures(, ...)`

CopyLBearing
`CopyLBearing()`

CopyRBearing
`CopyRBearing()`

CopyReference
`CopyReference()`

CopyUnlinked
`CopyUnlinked()`

CopyVWidth
`CopyVWidth()`

CopyWidth
`CopyWidth()`

CorrectDirection
`CorrectDirection(`*arg*`)`

Cos
`Cos(`*theta*`)`

Returns the cosine of *theta*, which is measured in radians. This function may be used without a loaded font.

Cut
`Cut()`

DebugCrashFontForge
`DebugCrashFontForge(...)`

Causes FontAnvil [sic] to attempt to write to a null pointer, which should crash the interpreter. This may be of some use in debugging. Arguments are ignored. Function name retained for compatibility. In FontForge, this function requires a loaded font, but that limitation is removed in FontAnvil.

DefaultOtherSubrs
`DefaultOtherSubrs()`

Set the global store of "other subroutines," which are fragments of PostScript code used for special purposes within PostScript fonts, to its default contents specified by Adobe. This function may be used without a loaded font.

DefaultRoundToGrid
`DefaultRoundToGrid()`

DefaultUseMyMetrics
`DefaultUseMyMetrics()`

DetachAndRemoveGlyphs
`DetachAndRemoveGlyphs(, ...)`

DetachGlyphs
`DetachGlyphs(, ...)`

DontAutoHint
`DontAutoHint()`

DrawsSomething
`DrawsSomething(`*arg*`)`

Error
`Error(`*msg*`)`

This function may be used without a loaded font.

Exp
`Exp(`x`)`

Compute the exponential function, e^x. This function may be used without a loaded font.

ExpandStroke
`ExpandStroke(`*arg*, *arg*, [*arg*], [*arg*], [*arg*], [*arg*]`)`

Export
`Export(`*arg*, *arg*`)`

FileAccess
`FileAccess(`*arg*, *arg*`)`

This function may be used without a loaded font.

FindIntersections
`FindIntersections()`

FindOrAddCvtIndex
`FindOrAddCvtIndex(`*arg*, *arg*`)`

Floor
`Floor(`*arg*`)`

Returns $\lfloor x \rfloor$. That is the floor of x, or the greatest integer less than or equal to x. This function may be used without a loaded font.

FontImage
`FontImage(`*arg*, *arg*, *arg*, [*arg*]`)`

FontsInFile
`FontsInFile(`*arg*`)`

This function may be used without a loaded font.

Generate
`Generate(`*arg*, *arg*, [*arg*], [*arg*], [*arg*], [*arg*]`)`

GenerateFamily
`GenerateFamily(`*arg*, *arg*, *arg*, *arg*`)`

GenerateFeatureFile
`GenerateFeatureFile(`*arg*, *arg*`)`

GetAnchorPoints
`GetAnchorPoints(, ...)`

GetCoverageCounts
`GetCoverageCounts()`

Print (to standard output) a table listing all the built-in functions in the interpreter and how many times each one has been called in the current run of FontAnvil; this information may be useful in verifying the coverage of an interpreter test suite. This function may be used without a loaded font. This function is a FontAnvil extension and is not available in FontForge. The details of its operation may change in some future version. *ff*

GetCvtAt
`GetCvtAt(`*arg*`)`

GetEnv
`GetEnv(`*arg*`)`

Return the string value of an environment variable, or an empty string if the variable requested does not exist. This function may be used without a loaded font. In FontForge, requesting a non-existent variable is an error. *ff*

GetFontBoundingBox
`GetFontBoundingBox()`

GetLookupInfo
`GetLookupInfo(`*arg*`)`

GetLookupOfSubtable
`GetLookupOfSubtable(`*arg*`)`

GetLookupSubtables
`GetLookupSubtables(`*arg*`)`

GetLookups
`GetLookups(`*arg*`)`

GetMaxpValue

`GetMaxpValue(`*arg*`)`

GetOS2Value

`GetOS2Value(, ...)`

GetPosSub

`GetPosSub(`*arg*`)`

GetPref

`GetPref(`*arg*`)`

This function may be used without a loaded font.

GetPrivateEntry

`GetPrivateEntry(`*arg*`)`

GetSubtableOfAnchorClass

`GetSubtableOfAnchorClass(`*arg*`)`

GetTTFName

`GetTTFName(`*arg*`, `*arg*`)`

GetTeXParam

`GetTeXParam(`*arg*`)`

GlyphInfo

`GlyphInfo(, ...)`

HFlip

`HFlip(`*arg*`)`

HasPreservedTable

`HasPreservedTable(`*arg*`)`

HasPrivateEntry

`HasPrivateEntry(`*arg*`)`

Import

`Import(`*arg*`, `*arg*`, [`*arg*`])`

InFont

`InFont([`*arg*`])`

Inline

`Inline(`*arg*`, `*arg*`)`

Int

`Int(`x`)`

Converts a real number or Unicode code point value to an integer, by means of the C compiler's type coercion. In the case of real numbers, that means x will be rounded toward zero. An integer argument will be returned unchanged. This function may be used without a loaded font.

InterpolateFonts

`InterpolateFonts(`*arg*`, `*arg*`, `*arg*`)`

IsAlNum

`IsAlNum(`*arg*`)`

This function may be used without a loaded font.

IsAlpha

`IsAlpha(`*arg*`)`

This function may be used without a loaded font.

IsDigit

`IsDigit(`*arg*`)`

This function may be used without a loaded font.

IsFinite

`IsFinite(`*arg*`)`

This function may be used without a loaded font.

IsHexDigit

`IsHexDigit(`*arg*`)`

This function may be used without a loaded font.

IsLower

`IsLower(`*arg*`)`

This function may be used without a loaded font.

IsNan

`IsNan(`*arg*`)`

This function may be used without a loaded font.

IsSpace

`IsSpace(`*arg*`)`

This function may be used without a loaded font.

IsUpper

`IsUpper(`*arg*`)`

This function may be used without a loaded font.

Italic

`Italic([`*arg*`], [`*arg*`], [`*arg*`], [`*arg*`], [`*arg*`], [`*arg*`], [`*arg*`], [`*arg*`], [`*arg*`])`

Join

`Join()`

LoadEncodingFile

`LoadEncodingFile(`*arg*, *arg*`)`

This function may be used without a loaded font.

LoadNamelist

`LoadNamelist(`*arg*`)`

This function may be used without a loaded font.

LoadNamelistDir

`LoadNamelistDir(`[*arg*]`)`

This function may be used without a loaded font.

LoadStringFromFile

`LoadStringFromFile(`*arg*`)`

This function may be used without a loaded font.

LoadTableFromFile

`LoadTableFromFile(`*arg*, *arg*`)`

Log

`Log(`x`)`

Returns $\ln x$, that is the natural (base e) logarithm. This is an error if x is zero, negative, or not a number. This function may be used without a loaded font.

LookupStoreLigatureInAfm

`LookupStoreLigatureInAfm(`*arg*, *arg*`)`

MMAxisBounds

`MMAxisBounds(`*arg*`)`

MMAxisNames

`MMAxisNames()`

MMBlendToNewFont

`MMBlendToNewFont(, ...)`

MMChangeInstance

`MMChangeInstance(`*arg*`)`

MMChangeWeight

`MMChangeWeight(, ...)`

MMInstanceNames

`MMInstanceNames()`

MMWeightedName

`MMWeightedName()`

MakeLine

`MakeLine(, ...)`

MergeFeature

`MergeFeature(`*arg*`)`

MergeFonts

`MergeFonts(`*arg*, *arg*`)`

MergeKern

`MergeKern(`*arg*`)`

MergeLookupSubtables

`MergeLookupSubtables(`*arg*, *arg*`)`

MergeLookups

`MergeLookups(`*arg*, *arg*`)`

Move

`Move(`*arg*, *arg*`)`

MoveReference

`MoveReference(`*arg*, *arg*, *arg*, ...`)`

MultipleEncodingsToReferences

`MultipleEncodingsToReferences()`

NameFromUnicode

`NameFromUnicode(`*arg*, *arg*`)`

This function may be used without a loaded font.

NearlyHvCps

`NearlyHvCps(`[*arg*], [*arg*]`)`

NearlyHvLines

`NearlyHvLines(`[*arg*]`)`

NearlyLines

`NearlyLines(`[*arg*]`)`

New

`New()`

This function may be used without a loaded font.

NonLinearTransform

`NonLinearTransform(`*xexp*, *yexp*`)`

Transform all the X and Y coordinates in the selected glyphs according to two expressions given as strings. The expressions are defined in a language unique to this function, which can be summarized

as follows, from highest to lowest precedence class, with evaluation from left to right within a class:

- constant reals, constant integers, and the variables `x` and `y`
- `!`, `()`, `sin()`, `cos()`, `tan()`, `log()`, `exp()`, `sqrt()`, `abs()`, `rint()`, `floor()`, `ceil()`
- `^`
- `*`, `/`, `%`
- `+`, `-`
- `==`, `!=`, `>=`, `>`, `<=`, `<`, `<`
- `&&`, `||`
- `? :` (C-style ternary question mark)

The FontForge code base supports this function, but only as a compile-time option that is turned off by default. Thus, most actual installations of FontForge will not support it. It is unconditionally enabled in FontAnvil. FontForge documents `x` and `y` as being the only atomic values available in the `NonLinearTransform()` expression language (no integers or reals), but that is obviously incorrect. FontForge also incorrectly documents `float()` as a function available in the language, instead of `floor()`.

Open

`Open(`*arg*, *arg*`)`

This function may be used without a loaded font.

Ord

`Ord(`*str*`)`

Converts a string to an array of integers in the range 1 to 255, representing the byte values in the string. Note 0 is not included because PE Script strings cannot contain zero bytes. This is the inverse of the `Chr()` function; see also `Utf8()` and `Ucs4()`, which operate on Unicode strings and code points instead of byte values.

`Ord(`*str*, *pos*`)`

With the optional *pos* argument, `Ord()` returns a single integer instead of an array, representing the byte value at the given zero-based index into the string.

This function may be used without a loaded font.

Outline

`Outline(`*arg*`)`

OverlapIntersect

`OverlapIntersect()`

Paste

`Paste()`

PasteInto

`PasteInto()`

PasteWithOffset

`PasteWithOffset(`*arg*, *arg*`)`

PositionReference

`PositionReference(`*arg*, *arg*, *arg*, ...`)`

PostNotice

`PostNotice(`*arg*`)`

This function may be used without a loaded font.

Pow

`Pow(`*arg*, *arg*`)`

This function may be used without a loaded font.

PreloadCidmap

`PreloadCidmap(`*arg*, *arg*, *arg*, *arg*`)`

This function may be used without a loaded font.

Print

`Print(`, ...`)`

This function may be used without a loaded font.

PrintFont

`PrintFont(`*arg*, *arg*, [*arg*], [*arg*]`)`

PrintSetup

`PrintSetup(`*arg*, *arg*, [*arg*], [*arg*]`)`

This function may be used without a loaded font.

PrivateGuess

`PrivateGuess(`*arg*`)`

Quit

`Quit(`[*arg*]`)`

This function may be used without a loaded font.

Rand

`Rand()`

Return something called "a random integer." FontForge does not document what that means. In fact, in FontForge it means whatever comes out of a call to the C library `rand()` function, and what that actually is will vary depending on the host platform. FontAnvil uses a bundled copy of the SIMD-oriented Fast Mersenne Twister by Saito and

Matsumoto. The return values are uniformly distributed over the integers 0 to $2^{31} - 1$, that is 0 to 2147483647 (32-bit integers from the generator with the high bit forced to zero to prevent signedness problems). The generator is seeded from the system clock and process ID when the interpreter starts. The same generator instance is used throughout FontAnvil wherever random numbers are called for, including in operations that do not obviously involve randomization (as a result of UUID generation and such). Thus, scripts should not depend on its producing a consistent sequence of numbers.

This function may be used without a loaded font. See `RandReal()` for a related function that may be of use.

RandReal

`RandReal()`

Return a pseudorandom real number in the interval $[0, 1)$. This uses the same underlying generator as `Rand()`, which see; but its output range may be more convenient. In principle, `RandReal()` may also have slightly better precision, because it uses all 32 bits of the PRNG output. This function is a FontAnvil extension and is not available in FontForge. This function may be used without a loaded font.

ReadOtherSubrsFile

`ReadOtherSubrsFile(`*arg*`)`

This function may be used without a loaded font.

Real

`Real(`*arg*`)`

This function may be used without a loaded font.

Reencode

`Reencode(`*arg*, *arg*`)`

RemoveAllKerns

`RemoveAllKerns()`

RemoveAllVKerns

`RemoveAllVKerns()`

RemoveAnchorClass

`RemoveAnchorClass(`*arg*`)`

RemoveDetachedGlyphs

`RemoveDetachedGlyphs(, ...)`

RemoveLookup

`RemoveLookup(`*arg*, *arg*`)`

RemoveLookupSubtable

`RemoveLookupSubtable(`*arg*, *arg*`)`

RemoveOverlap

`RemoveOverlap()`

RemovePosSub

`RemovePosSub(`*arg*`)`

RemovePreservedTable

`RemovePreservedTable(`*arg*`)`

RenameGlyphs

`RenameGlyphs(`*arg*`)`

ReplaceCharCounterMasks

`ReplaceCharCounterMasks(`*arg*`)`

ReplaceCvtAt

`ReplaceCvtAt(`*arg*, *arg*`)`

ReplaceGlyphCounterMasks

`ReplaceGlyphCounterMasks(`*arg*`)`

ReplaceWithReference

`ReplaceWithReference(`[*arg*], [*arg*]`)`

Revert

`Revert()`

Reload the current font from its file on disk (which must exist).

RevertToBackup

`RevertToBackup()`

Reload the current font from its backup file (the one ending in ~, not a numbered revision), if one exists. This is only meaningful for fonts that are SFD files for which a backup was created by a previous `Save()`.

Rotate

`Rotate(`*arg*, [*arg*], [*arg*]`)`

Round

`Round(`*arg*`)`

Returns the nearest integer to x, with ties broken by returning the nearest *even* integer. This behaviour may differ on nonstandard systems that

lack the `fesetround()` library function. This function may be used without a loaded font.

RoundToCluster
`RoundToCluster(`[*arg*], [*arg*]`)`

RoundToInt
`RoundToInt(`*arg*`)`

Round coordinates in selected glyphs. See `Round()` for rounding a real number to an integer.

SameGlyphAs
`SameGlyphAs()`

Save
`Save(`[*arg*], [*arg*]`)`

SaveTableToFile
`SaveTableToFile(`*arg*, *arg*`)`

Scale
`Scale(`*arg*, *arg*, [*arg*], [*arg*]`)`

ScaleToEm
`ScaleToEm(`*arg*, *arg*`)`

Select
`Select(, ...)`

SelectAll
`SelectAll()`

SelectAllInstancesOf
`SelectAllInstancesOf(, ...)`

SelectBitmap
`SelectBitmap(`*arg*`)`

SelectByColor
`SelectByColor(`*arg*`)`

SelectByColour
`SelectByColour(`*arg*`)`

SelectByPosSub
`SelectByPosSub(`*arg*, *arg*`)`

SelectChanged
`SelectChanged(`*arg*`)`

SelectFewer
`SelectFewer(`*arg*, ...`)`

SelectFewerSingletons
`SelectFewerSingletons(`*arg*, ...`)`

SelectGlyphsBoth
`SelectGlyphsBoth(`*arg*`)`

SelectGlyphsReferences
`SelectGlyphsReferences(`*arg*`)`

SelectGlyphsSplines
`SelectGlyphsSplines(`*arg*`)`

SelectHintingNeeded
`SelectHintingNeeded(`*arg*`)`

SelectIf
`SelectIf(, ...)`

SelectInvert
`SelectInvert()`

SelectMore
`SelectMore(`*arg*, ...`)`

SelectMoreIf
`SelectMoreIf(`*arg*, ...`)`

SelectMoreSingletons
`SelectMoreSingletons(`*arg*, ...`)`

SelectMoreSingletonsIf
`SelectMoreSingletonsIf(`*arg*, ...`)`

SelectNone
`SelectNone()`

SelectSingletons
`SelectSingletons(, ...)`

SelectSingletonsIf
`SelectSingletonsIf(, ...)`

SelectWorthOutputting
`SelectWorthOutputting(`*arg*`)`

SetCharCnt
`SetCharCnt(`*arg*`)`

SetCharColor
`SetCharColor(`*arg*`)`

SetCharComment
`SetCharComment(`*arg*`)`

SetCharCounterMask
`SetCharCounterMask(`*arg*, *arg*, ...`)`

SetCharName
`SetCharName(`*arg*, *arg*`)`

SetFeatureList
`SetFeatureList(`*arg*, *arg*`)`

SetFondName
`SetFondName(`*arg*`)`

SetFontHasVerticalMetrics
`SetFontHasVerticalMetrics(`*arg*`)`

SetFontNames
`SetFontNames(`*arg*, *arg*, [*arg*], [*arg*], [*arg*], [*arg*]`)`

SetFontOrder
`SetFontOrder(`*arg*`)`

SetGasp
`SetGasp(`, ...`)`

SetGlyphChanged
`SetGlyphChanged(`*arg*`)`

SetGlyphClass
`SetGlyphClass(`*arg*`)`

SetGlyphColor
`SetGlyphColor(`*arg*`)`

SetGlyphComment
`SetGlyphComment(`*arg*`)`

SetGlyphCounterMask
`SetGlyphCounterMask(`, ...`)`

SetGlyphName
`SetGlyphName(`, ...`)`

SetGlyphTeX
`SetGlyphTeX(`*arg*, *arg*, *arg*, [*arg*]`)`

SetItalicAngle
`SetItalicAngle(`*arg*, *arg*`)`

SetKern
`SetKern(`*arg*, *arg*, *arg*`)`

SetLBearing
`SetLBearing(`*arg*, *arg*`)`

SetMacStyle
`SetMacStyle(`*arg*`)`

SetMaxpValue
`SetMaxpValue(`*arg*, *arg*`)`

SetOS2Value
`SetOS2Value(`, ...`)`

SetPanose
`SetPanose(`*arg*, *arg*`)`

SetPref
`SetPref(`*arg*, *arg*, *arg*`)`

This function may be used without a loaded font.

SetRBearing
`SetRBearing(`*arg*, *arg*`)`

SetTTFName
`SetTTFName(`*arg*, *arg*, *arg*`)`

SetTeXParams
`SetTeXParams(`, ...`)`

SetUnicodeValue
`SetUnicodeValue(`*arg*, *arg*`)`

SetUniqueID
`SetUniqueID(`*arg*`)`

SetVKern
`SetVKern(`*arg*, *arg*, *arg*`)`

SetVWidth
`SetVWidth(`*arg*, *arg*`)`

SetWidth
`SetWidth(`*arg*, *arg*`)`

Shadow
`Shadow(`*arg*, *arg*, *arg*`)`

Shell

`Shell(`*command*`)`

Execute a command in the system shell, via the C library `system(3)` call. Returns the integer return value from doing so. All the usual security considerations associated with shell interfaces apply. This function may be used without a loaded font. This function is a FontAnvil extension and is not available in FontForge.

ff

Simplify

`Simplify(, ...)`

Sin

`Sin(`*theta*`)`

Returns the sine of *theta*, which is measured in radians. This function may be used without a loaded font.

SizeOf

`SizeOf(`*arg*`)`

This function may be used without a loaded font.

Skew

`Skew(`*arg*, *arg*, [*arg*], [*arg*]`)`

SmallCaps

`SmallCaps(`[*arg*], [*arg*], [*arg*], [*arg*]`)`

Sqrt

`Sqrt(`*arg*`)`

This function may be used without a loaded font.

StrJoin

`StrJoin(`*arg*, *arg*`)`

This function may be used without a loaded font.

StrSplit

`StrSplit(`*arg*, *arg*, *arg*`)`

This function may be used without a loaded font.

Strcasecmp

`Strcasecmp(`*arg*, *arg*`)`

This function may be used without a loaded font.

Strcasestr

`Strcasestr(`*arg*, *arg*`)`

This function may be used without a loaded font.

Strftime

`Strftime(`*arg*, *arg*, [*arg*]`)`

This function may be used without a loaded font.

Strlen

`Strlen(`*arg*`)`

This function may be used without a loaded font.

Strrstr

`Strrstr(`*arg*, *arg*`)`

This function may be used without a loaded font.

Strskipint

`Strskipint(`*arg*, *arg*`)`

This function may be used without a loaded font.

Strstr

`Strstr(`*arg*, *arg*`)`

This function may be used without a loaded font.

Strsub

`Strsub(`*arg*, *arg*, *arg*`)`

This function may be used without a loaded font.

Strtod

`Strtod(`*arg*`)`

This function may be used without a loaded font.

Strtol

`Strtol(`*arg*, *arg*`)`

This function may be used without a loaded font.

SubstitutionPoints

`SubstitutionPoints()`

Tan

`Tan(`*theta*`)`

Returns the tangent of *theta*, which is measured in radians. This function may be used without a loaded font.

ToLower

`ToLower(`*arg*`)`

This function may be used without a loaded font.

ToMirror

`ToMirror(`*arg*`)`

This function may be used without a loaded font.

ToString

`ToString(`*arg*`)`

This function may be used without a loaded font.

ToUpper

`ToUpper(`*arg*`)`

This function may be used without a loaded font.

Transform

`Transform(`*arg*, *arg*, *arg*, *arg*, *arg*, *arg*`)`

TypeOf

`TypeOf(`*arg*`)`

This function may be used without a loaded font.

UCodePoint

`UCodePoint(`*int*`)`

Cast an integer to the special "Unicode code point" data type required by some other scripting functions. This function may be used without a loaded font.

Ucs4

`Ucs4(`*str*`)`

Decode a UTF-8 string into an array of integers (not actually UCS4, despite the name) expressing the code points in the string. Handling of illegal UTF-8, including encoded surrogate code points, is undefined. This is the inverse of the `Utf8()` function; see also `Chr()` and `Ord()`, which operate on byte values instead of code points. This function may be used without a loaded font.

UnicodeAnnotationFromLib

`UnicodeAnnotationFromLib(`*arg*`)`

This function may be used without a loaded font.

UnicodeBlockEndFromLib

`UnicodeBlockEndFromLib(`*arg*`)`

This function may be used without a loaded font.

UnicodeBlockNameFromLib

`UnicodeBlockNameFromLib(`*arg*`)`

This function may be used without a loaded font.

UnicodeBlockStartFromLib

`UnicodeBlockStartFromLib(`*arg*`)`

This function may be used without a loaded font.

UnicodeFromName

`UnicodeFromName(`*arg*`)`

This function may be used without a loaded font.

UnicodeNameFromLib

`UnicodeNameFromLib(`*arg*`)`

This function may be used without a loaded font.

UnicodeNamesListVersion

`UnicodeNamesListVersion()`

This function may be used without a loaded font.

UnlinkReference

`UnlinkReference()`

Utf8

`Utf8(`*codes*`)`

Encode an array of integers in the range 0 to 0x10FFFF, or a such single integer, into a UTF-8 string. Surrogate code point values will result in an illegal UTF-8 result, and zeroes will prematurely terminate the output. This is the inverse of the `Ucs4()` function; see also `Chr()` and `Ord()`, which operate on byte values instead of code points. This function may be used without a loaded font.

VFlip

`VFlip(`*arg*`)`

VKernFromHKern

`VKernFromHKern()`

Validate

`Validate([`*arg*`])`

Wireframe

`Wireframe(`*arg*, *arg*, *arg*`)`

WorthOutputting

`WorthOutputting(`*arg*`)`

WriteStringToFile

`WriteStringToFile(`*arg*, *arg*, *arg*`)`

This function may be used without a loaded font.

WritePfm

`WritePfm(`*arg*`)`

Built-in functions in FontForge and not in FontAnvil

`LoadPlugin()`
`LoadPluginDir()` *ff*

Removed because FontAnvil does not use plug-ins.

`LoadPrefs()`
`SavePrefs()` *ff*

Removed because, to ensure consistent results from scripts, FontAnvil does not store per-user "preferences" between runs. For the moment, other functions related to FontForge-style preference variables remain in the language, but the values of these variables are initialized to defaults at the start of each script run.

ff
`PrintFont()`
`PrintSetup()`

Removed because of the grossly disproportionate amount of unportable interfacing code needed to talk to the native printing interfaces on each operating system. Some future version may support a more portable printing feature, likely involving writing to files instead of interfacing directly to the printer drivers.

ff
`Autotrace()`
`bAutoCounter()`
`bDontAutoHint()`
`bSubstitutionPoints()`
`BuildComposit()`
`GetPrefs()`

Misspellings of documented function names which FontForge once supported by mistake and now retains for compatibility with hypothetical deployed scripts that might have depended upon them. No such deployed scripts are actually known to exist. FontAnvil does not retain the misspelled names.

ff
`AddATT()`
`DefaultATT()`
`PrivateToCvt()`
`RemoveATT()`
`SelectByATT()`

Deprecated functions that only produce error messages in FontForge; removed entirely in FontAnvil.

Built-in variables

Licensing

George Williams put most of his work on PfaEdit, and subsequently FontForge, under licensing notices such as this one:

> Copyright © [years] by George Williams
>
> Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
>
> Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
>
> Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
>
> The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.
>
> THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

That is what is commonly called a *Three-Clause BSD License*. There were many subsequent contributors to the software (see the `AUTHORS` file in the root of a distribution tarball or version control checkout) and many of them were content to keep the same license terms in place, with or without adding their own names and years to the copyright notice at the top.

However, some contributors have placed additional restrictions on their work, most notably *GNU GPL3+* licenses like this one:

> Copyright © [year] [contributor's name]
>
> This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses/.

The Free Software Foundation takes the position that the three-clause BSD license is GPL-compatible,* meaning that it is legally permissible for a package that is under GPL3+ as a whole to include material

*http://www.gnu.org/licenses/license-list.html#ModifiedBSD

that is under three-clause BSD. The presence of GPL3+ contributions, however, forces the package as a whole to be licensed GPL3+. That being the case, both FontForge and FontAnvil should be treated as GPL3+, just with the awareness that some files may also be used separately from the package under the less restrictive three-clause BSD license.

It is the current practice of the FontForge project to encourage contributors of new material to apply GPL3+ notices to any new files, but retain the BSD notices on files that already have those. There was an incident in which someone tried to apply a patch someone else had written to a currently BSD-licensed file in FontForge—with the patch to become roughly 1/1000th of the file's total volume. The author of the patch demanded that the whole file should become GPL3+, overriding the BSD notice on it and the apparent intentions of the previous contributors to that file. I would like to avoid such incidents.

A few files have other distribution terms. In particular, some parts of the build system have very permissive licenses.

FontForge attempts to maintain a list of all the licensing terms of all the files in the project; but their list has never been up to date, cannot reasonably be expected to ever be up to date nor to stay up to date even if it ever is at one moment, currently contains incorrect information, and seems unnecessary. I do not propose to make such a list for FontAnvil.

The current licensing policy for FontAnvil is substantially the same as that of FontForge:

- FontAnvil as a whole is covered by the GPL version 3, or any later version.
- Some files in FontAnvil are also available under less restrictive licenses. You must consult the notices in those files for details.
- I will place GPL3+ notices on new files I create within FontAnvil, and encourage others to do the same.
- I will leave files with existing broader-than-GPL notices under their existing notices (possibly adding my own name and year copyright lines), and encourage others to do the same.
- I will not accept contributions that entail drastic changes to the licensing status of work done by persons other than the contributor, and I will discourage the submission of such contributions.

Finally, note that although FontAnvil is associated with the Tsukurimashou Project, its licensing is not identical to that of other things included in the Tsukurimashou Project.

Index

AddAccent(), 19
AddAnchorClass(), 19
AddAnchorPoint(), 19
AddDHint(), 19
AddExtrema(), 20
AddHHint(), 20
AddInstrs(), 20
AddLookup(), 20
AddLookupSubtable(), 20
AddPosSub(), 20
AddSizeFeature(), 20
AddVHint(), 20
ApplySubstitution(), 20
Array(), 20
AskUser(), 21
ATan2(), 19
AutoCounter(), 21
AutoHint(), 21
AutoInstr(), 21
AutoKern(), 21
AutoTrace(), 21
AutoWidth(), 21

BitmapsAvail(), 21
BitmapsRegen(), 21
BuildAccented(), 21
BuildComposite(), 21
BuildDuplicate(), 21

CanonicalContours(), 22
CanonicalStart(), 22
Ceil(), 22
CenterInWidth(), 22
ChangePrivateEntry(), 22
ChangeWeight(), 22
CharCnt(), 22
CharInfo(), 22
CheckForAnchorClass(), 22
Chr(), 22
CIDChangeSubFont(), 22
CIDFlatten(), 22
CIDFlattenByCMap(), 22
CIDSetFontNames(), 22
Clear(), 22
ClearBackground(), 22
ClearCharCounterMasks(), 22
ClearGlyphCounterMasks(), 23
ClearHints(), 23
ClearInstrs(), 23
ClearPrivateEntry(), 23
ClearTable(), 23
Close(), 23
CompareFonts(), 23
CompareGlyphs(), 23
ControlAfmLigatureOutput(), 23
ConvertByCMap(), 23
ConvertToCID(), 23
Copy(), 23
CopyAnchors(), 23
CopyFgToBg(), 23
CopyGlyphFeatures(), 23
CopyLBearing(), 23
CopyRBearing(), 23
CopyReference(), 23
CopyUnlinked(), 23
CopyVWidth(), 23
CopyWidth(), 23
CorrectDirection(), 23
Cos(), 23
Cut(), 23

DebugCrashFontForge(), 23
DefaultOtherSubrs(), 23
DefaultRoundToGrid(), 23
DefaultUseMyMetrics(), 23
DetachAndRemoveGlyphs(), 23
DetachGlyphs(), 24
DontAutoHint(), 24
DrawsSomething(), 24

Error(), 24
Exp(), 24
ExpandStroke(), 24
Export(), 24

FileAccess(), 24
FindIntersections(), 24

FindOrAddCvtIndex(), 24
Floor(), 24
FontImage(), 24
FontsInFile(), 24

Generate(), 24
GenerateFamily(), 24
GenerateFeatureFile(), 24
GetAnchorPoints(), 24
GetCoverageCounts(), 24
GetCvtAt(), 24
GetEnv(), 24
GetFontBoundingBox(), 24
GetLookupInfo(), 24
GetLookupOfSubtable(), 24
GetLookups(), 24
GetLookupSubtables(), 24
GetMaxpValue(), 24
GetOS2Value(), 25
GetPosSub(), 25
GetPref(), 25
GetPrivateEntry(), 25
GetSubtableOfAnchorClass(), 25
GetTeXParam(), 25
GetTTFName(), 25
GlyphInfo(), 25

HasPreservedTable(), 25
HasPrivateEntry(), 25
HFlip(), 25

Import(), 25
InFont(), 25
Inline(), 25
Int(), 25
InterpolateFonts(), 25
IsAlNum(), 25
IsAlpha(), 25
IsDigit(), 25
IsFinite(), 25
IsHexDigit(), 25
IsLower(), 25
IsNan(), 25
IsSpace(), 25
IsUpper(), 25
Italic(), 25

Join(), 25

LoadEncodingFile(), 26
LoadNamelist(), 26
LoadNamelistDir(), 26
LoadStringFromFile(), 26
LoadTableFromFile(), 26
Log(), 26
LookupStoreLigatureInAfm(), 26

MakeLine(), 26
MergeFeature(), 26
MergeFonts(), 26
MergeKern(), 26
MergeLookups(), 26
MergeLookupSubtables(), 26
MMAxisBounds(), 26
MMAxisNames(), 26
MMBlendToNewFont(), 26
MMChangeInstance(), 26
MMChangeWeight(), 26
MMInstanceNames(), 26
MMWeightedName(), 26
Move(), 26
MoveReference(), 26
MultipleEncodingsToReferences(), 26

NameFromUnicode(), 26
NearlyHvCps(), 26
NearlyHvLines(), 26
NearlyLines(), 26
New(), 26
NonLinearTransform(), 26

Open(), 27
Ord(), 27
Outline(), 27
OverlapIntersect(), 27

Paste(), 27
PasteInto(), 27
PasteWithOffset(), 27
PositionReference(), 27
PostNotice(), 27
Pow(), 27
PreloadCidmap(), 27
Print(), 27
PrintFont(), 27
PrintSetup(), 27
PrivateGuess(), 27

Quit(), 27

Rand(), 27
RandReal(), 28
ReadOtherSubrsFile(), 28
Real(), 28

Reencode(), 28
RemoveAllKerns(), 28
RemoveAllVKerns(), 28
RemoveAnchorClass(), 28
RemoveDetachedGlyphs(), 28
RemoveLookup(), 28
RemoveLookupSubtable(), 28
RemoveOverlap(), 28
RemovePosSub(), 28
RemovePreservedTable(), 28
RenameGlyphs(), 28
ReplaceCharCounterMasks(), 28
ReplaceCvtAt(), 28
ReplaceGlyphCounterMasks(), 28
ReplaceWithReference(), 28
Revert(), 28
RevertToBackup(), 28
Rotate(), 28
Round(), 28
RoundToCluster(), 29
RoundToInt(), 29

SameGlyphAs(), 29
Save(), 29
SaveTableToFile(), 29
Scale(), 29
ScaleToEm(), 29
Select(), 29
SelectAll(), 29
SelectAllInstancesOf(), 29
SelectBitmap(), 29
SelectByColor(), 29
SelectByColour(), 29
SelectByPosSub(), 29
SelectChanged(), 29
SelectFewer(), 29
SelectFewerSingletons(), 29
SelectGlyphsBoth(), 29
SelectGlyphsReferences(), 29
SelectGlyphsSplines(), 29
SelectHintingNeeded(), 29
SelectIf(), 29
SelectInvert(), 29
SelectMore(), 29
SelectMoreIf(), 29
SelectMoreSingletons(), 29
SelectMoreSingletonsIf(), 29
SelectNone(), 29
SelectSingletons(), 29
SelectSingletonsIf(), 29
SelectWorthOutputting(), 29
SetCharCnt(), 29
SetCharColor(), 30
SetCharComment(), 30
SetCharCounterMask(), 30
SetCharName(), 30
SetFeatureList(), 30
SetFondName(), 30
SetFontHasVerticalMetrics(), 30
SetFontNames(), 30
SetFontOrder(), 30
SetGasp(), 30
SetGlyphChanged(), 30
SetGlyphClass(), 30
SetGlyphColor(), 30
SetGlyphComment(), 30
SetGlyphCounterMask(), 30
SetGlyphName(), 30
SetGlyphTeX(), 30
SetItalicAngle(), 30
SetKern(), 30
SetLBearing(), 30
SetMacStyle(), 30
SetMaxpValue(), 30
SetOS2Value(), 30
SetPanose(), 30
SetPref(), 30
SetRBearing(), 30
SetTeXParams(), 30
SetTTFName(), 30
SetUnicodeValue(), 30
SetUniqueID(), 30
SetVKern(), 30
SetVWidth(), 30
SetWidth(), 30
Shadow(), 30
Shell(), 31
Simplify(), 31
Sin(), 31
SizeOf(), 31
Skew(), 31
SmallCaps(), 31
Sqrt(), 31
Strcasecmp(), 31
Strcasestr(), 31
Strftime(), 31
StrJoin(), 31
Strlen(), 31
Strrstr(), 31
Strskipint(), 31
StrSplit(), 31
Strstr(), 31

Strsub(), 31
Strtod(), 31
Strtol(), 31
SubstitutionPoints(), 31

Tan(), 31
ToLower(), 31
ToMirror(), 31
ToString(), 31
ToUpper(), 32
Transform(), 32
TypeOf(), 32

UCodePoint(), 32
Ucs4(), 32
UnicodeAnnotationFromLib(), 32
UnicodeBlockEndFromLib(), 32
UnicodeBlockNameFromLib(), 32
UnicodeBlockStartFromLib(), 32
UnicodeFromName(), 32
UnicodeNameFromLib(), 32
UnicodeNamesListVersion(), 32
UnlinkReference(), 32
Utf8(), 32

Validate(), 32
VFlip(), 32
VKernFromHKern(), 32

Wireframe(), 32
WorthOutputting(), 32
WritePfm(), 32
WriteStringToFile(), 32